Tailoring Quantum Architectures to Implementation Style: A Quantum Computer for Mobile and Persistent Qubits

Eric Chi, Stephen A. Lyon, Margaret Martonosi
Dept. of Electrical Engineering, Princeton University
{echi,lyon,mrm}@princeton.edu

ABSTRACT

In recent years, quantum computing (QC) research has moved from the realm of theoretical physics and mathematics into real implementations [9]. With many different potential hardware implementations, quantum computer architecture is a rich field with an opportunity to solve interesting new problems and to revisit old ones. This paper presents a QC architecture tailored to physical implementations with highly mobile and persistent quantum bits (qubits). Implementations with qubit coherency times that are much longer than operation times and qubit transportation times that are orders of magnitude faster than operation times lend greater flexibility to the architecture. This is particularly true in the placement and locality of individual qubits. For concreteness, we assume a physical device model based on electron-spin qubits on liquid helium (eSHe) [15].

Like many conventional computer architectures, QCs focus on the efficient exposure of parallelism. We present here a QC microarchitecture that enjoys increasing computational parallelism with size and latency scaling only linearly with the number of operations. Although an efficient and high level of parallelism is admirable, quantum hardware is still expensive and difficult to build, so we demonstrate how the software may be optimized to reduce an application's hardware requirements by 25% with no performance loss. Because the majority of a QC's time and resources are devoted to quantum error correction, we also present noise modeling results that evaluate error correction procedures. These results demonstrate that idle qubits in memory need only be refreshed approximately once every one hundred operation cycles.

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Other Architecture Styles; C.2.1 [Computer-Communication Networks]: Network Architecture and Design

General Terms: Design

Keywords: architecture, quantum

1. INTRODUCTION

Quantum computing is an exciting new computing paradigm that offers the opportunity for exponential speedup over classical computation. Physicists have proposed many different implementations for realizing quantum bits (qubits) and operations. Quantum computer (QC) implementations share many common characteristics; most notably, all physical implementations must contend with *decoherence*: the accumulation of noise-induced errors over time and via imperfect operations. However, QC implementations vary significantly in many parameters, including the susceptibility of qubits to decoherence and the relative speeds of qubit communication and operation. These varying parameters lead to QC architectures with starkly different design emphases.

Meaningful quantum computation necessarily involves operations acting on multiple qubits, and quantum algorithms are designed assuming the QC performs many of these operations in parallel. Because multiple-qubit operations require their operand qubits to be physically proximate, QC architectures must be designed to efficiently transport a large number of operand qubits in order to maximize parallel execution.

Previously proposed QC architectures have been designed for implementation technologies where communication costs are significantly greater than computation costs [12, 13]. Such architectures expend many resources to overlap communication with execution time to emphasize locality in qubit operations. *Quantum teleportation* has been proposed as the long-distance communication solution for such architectures [20]. Instead of physically transporting an operand qubit, quantum teleportation moves two specially prepared qubits (an EPR pair) to form the endpoints of a single-use communication channel [2]. On-chip communication via quantum teleportation is contingent on a chip-wide network that prepares, purifies [3], and distributes EPR pairs to manage these communication channels.

This paper examines an alternate implementation style for QCs. Our architecture design approach is based on different technology assumptions for implementations in which transportation times are significantly faster than operation times. Memory errors due to decoherence are infrequent, particularly when the qubit is not participating in an operation. QCs with highly mobile qubits and a stable memory rely less heavily on on locality, leading to simpler and more flexible architectures. When transportation time is much faster than operation time, quantum teleportation is rarely worthwhile, and the QC architecture no longer requires an EPR network. Our contributions are as follows:

- Building upon the electron-spins on helium (eSHe) QC implementation [15], we present a simple architecture that is capable of transporting a large number of operands in parallel to support a high level of execution parallelism. Our strategy is notable in that its space and performance costs scale only linearly with computation size.
- We present noise modeling results that evaluate error correction protocols and assess the robustness of our memory.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

- We also present compiler strategies for optimizing performance and hardware requirements.

Section 2 describes some of the basic mechanics required to understand architectural problems in quantum computing. We focus on a particular high-mobility QC implementation: the electron spins on liquid helium (eSHe) technology described in Section 3. Section 4 presents our computer architecture that builds upon the advantages and requirements of this implementation. Section 5 describes our approach to efficiently transporting a large number of operands in parallel that enables our architecture to perform well. Section 6 analyzes error correction routines with our noise simulator. Section 7 describes our compilation and optimization strategies. We conclude with related work and discussion.

2. QUANTUM COMPUTING BASICS

Quantum computing is a new computing paradigm that takes advantage of distinctive properties in quantum mechanics. Quantum superposition allows quantum bits (qubits) to represent multiple states simultaneously. Whereas a classical n-bit string may possess exactly one of 2^n possible values, a string of n qubits can represent a simultaneous superposition of all 2^n states. Operations on qubits affect all their superposition states simultaneously, and it is this quantum parallelism that gives quantum computers the potential for exponential speedup over classical computers. Integer factoring is an example of a difficult problem with no known polynomial solution on classical computers, but Shor's algorithm shows that it may be solved in polynomial time on a QC [26]. Factoring has an enormous practical application in defeating public-key cryptosystems.

Building and controlling quantum systems is extraordinarily difficult, however, and isolating qubits from the surrounding environment is a considerable challenge. The environment around a QC may include random electromagnetic waves that introduce noise into the qubit states. This decoherence of quantum state leads to unreliability and requires a fault-tolerant approach to computing. Quantum error correction has been developed to encode a *logical* qubit state as a *code block* composed of multiple *physical* qubits so that random errors affecting individual physical qubits may be diagnosed and corrected to maintain the logical state. In this paper we adopt Steane's [[7,1,3]] quantum error correcting code (QECC) [30] that employs a 7-bit code block to represent a single logical qubit.

Fault-tolerant (FT) quantum computing protocols have been developed to interleave computation and error correction steps in a manner that prevents random errors from propagating out of control [32] [22]. FT protocols execute a program's *logical* operations directly on an encoded logical qubit via multiple *physical* operations. This approach avoids faults during program operations; an error on an non-encoded operation would be uncorrectable, but an error during a logical operation may be remedied. After a logical operation has effected its desired state change, an error correction recovery process is applied to the operand qubit(s). This process is described in further detail in Section 6.

QC implementations provide a limited number of physical operations at the hardware level. The common conceivable set of physical operations include arbitrary 1-qubit operations and a limited set of 2-qubit operations, with controlled-Not (CNOT) and controlled-Phase (CPHASE) gates being the most frequently used. 2-qubit CNOTs and arbitrary 1-qubit operations form a universal set of quantum operators [18]; their composition spans the set of all possible quantum operations. Physical operations may need to be performed in special locations or operating zones (*opzones*), which necessitates transportation instructions in the hardware to control datapath usage. Furthermore, 2-qubit operations require the operand qubits to be adjacent to each other so that their quantum mechanical states may interact; such operations always imply a need to transport at least one operand qubit state.

Overall, a quantum application is built in three stages. First, quantum applications are programmed as a series of arbitrary operations acting on set of program qubits. Second, a translation layer decomposes these arbitrary program operations into a finite set of available fault-tolerant logical operations that act on encoded logical qubits. This layer also implements error correction routines. Third, the underlying hardware implements the FT logical operations and error recoveries via 1- and 2-qubit physical operations plus transportation instructions to move operand qubits.

Many different schemes have been proposed to implement the quantum bits and operations for a scalable quantum computer. For example, the ion-trap QC [27][34][29] represents qubits using the electronic or nuclear states of individual ions. These ions are suspended in a 2D array of traps interconnected via a quantum charge-coupled device (QCCD) [13]. The Kane quantum computer [12] has immobile qubits in the form of a 2D array of phosphorous donor ions precisely positioned in a silicon lattice.

Both of these implementations have transportation schemes that are significantly slower than their operation times. The ion trap QC has high transportation constant costs with its high splitting and turning times (10-20 ms compared to 1-34 μs for operations [29]). The Kane QC communicates its immobile qubits' states via successive SWAP operations on adjacent qubits yielding transportation time that is proportional to the number of hops travelled with every hop taking time similar to a 2-qubit physical operation. Both implementation technologies will likely rely on quantum teleportation for qubit state communication. However, teleportation incurs a substantial hardware cost in the increased number of qubits and number of operations required to purify EPR pairs and requires an extensive networking infrastructure to distribute EPR pairs throughout the chip. Purification has space costs that scale exponentially with the communication distance, and the preparation of each teleportation channel may require hundreds of EPR qubits [11].

Instead of an implementation where transportation time dominates operation time, this paper considers an architecture for an eSHe implementation. This technology is characterized by highly mobile qubits in which transportation time is expected to be much smaller than operating time. Fast transportation frees us from relying on quantum teleportation for long-range communication. Furthermore, eSHe architectures do not rely as heavily on spatial and temporal locality to minimize transportation time between operations.

3. ESHE DETAILS AND DEVICE MODEL

The eSHe QC uses the spins of individual electrons as the basic physical qubits. These electrons float in a vacuum above a layer of liquid helium, which provides a relatively noise-free environment. The eSHe approach is distinct from other electron qubits on helium proposals [6, 8, 21] which use charge qubits rather than spin qubits. The eSHe qubits will have a long memory time and are expected to maintain coherence when idle with an exponential decay constant of 100,000 seconds or nearly 28 hours [15][1].

[1] As with all QC technologies, eSHe characterizations of key noise and latency parameters continue to be refined experimentally [24]. Nonetheless it is informative to consider the architectural implications of eSHe, which lies in a very different design space from other promising QC schemes.

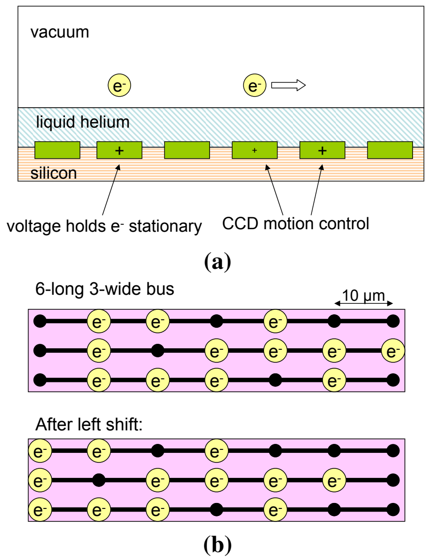


Figure 1: (a) A side view of the eSHe system (not to scale). Electron qubits float in a vacuum above a layer of liquid helium. Positively charged metal gates underneath the liquid helium attract these electrons and hold them stationary. The electrons can be moved via CCD control of these gates. (b) An example of a qubit transportation bus. Qubits move left or right along this bus akin to a shift register.

We can control the position of individual electrons by establishing an attractive positive voltage under every electron (Figure 1a). These positive potentials emanate from metal gates under the liquid helium layer. Qubits have great mobility as we can use these metal gates to shift the electrons' positions in the same manner as charged-coupled devices (CCD). Applying a voltage to an adjacent gate while reducing the voltage in a current gate encourages the electron to move and then hover over the adjacent gate. Unlike the ion trap scheme, which also utilizes CCD-style transportation, we do not expect eSHe qubits to have difficulty making turns because their motion is dampened by coupling with the surrounding gate electrodes [15]. The minimum separation distance between two qubits is $10\,\mu$m to avoid interacting their spin states. The CCD transportation speed is estimated to be 100 m/s. The CCD charge transfer efficiency has been measured to be at least 0.99999992 [23] and indicates a transfer failure rate of only once for every 300 m travelled.

In order to minimize control complexity, qubits travel along wire segments, and all qubits on a wire travel as a group, moving the same distance in the same direction. These wires are shift registers subject to the constraint that qubits may not be shifted past the endpoints of the wire (Figure 1b). Multiple wires may be tied together sharing a single shift control signal to form a bus. This shift register movement scheme is a SIMD-style transportation control mechanism and enables the movement of a large number of qubits with few control signals.

Operations are performed on the qubits via microwave pulses. Once again, to manage control complexity, we envision a SIMD approach. During operation, the operand qubits are situated in SIMD opzone columns, which is an array of execution units that operate in parallel. The opzones require a buffer distance between themselves and memory to avoid contaminating idle memory qubits with the microwave pulses. We estimate that a buffer distance of $100\,\mu$m will be sufficient (Figure 2). The opzones are designed to perform arbitrary single-qubit operations (with some limit on precision) and the CNOT and CPHASE 2-qubit operations.

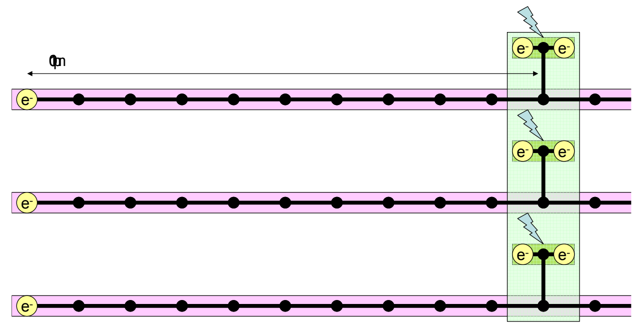


Figure 2: A 3-way opzone column performs SIMD operations. Idle qubits are separated from the opzones by at least $100\,\mu m$ to avoid microwave contamination.

movement speed	100 μm/μs
1-bit op time	1 μs
2-bit op time	1 ms
separation distance	10 μm
buffer distance surrounding operations	100 μm

Table 1: Device parameters for eSHe QC.

Single-qubit operations and measurements are estimated to be executed within 1 μs. Two-qubit operations are significantly more difficult and will take about 1 ms because of the weak magnetic dipole-dipole interaction. During the 2-qubit operation, the operand qubits will share the same opzone and will be situated in close proximity to one another to support the spin interaction needed for quantum operation. During 2-qubit operations, the qubit state is susceptible to significantly more decoherence, so their coherence decay constant is expected to drop to about 5,000 s. The infidelity of the 2-qubit operation itself is estimated at 10^{-4} due to precision timing requirements. We therefore anticipate 2-qubit operations to dominate the execution time and fidelity of any quantum application. Traveling qubits are estimated to have a decay constant of 25,000 s. With these parameters, a qubit may travel for 2.5 s (or 250 m) to accumulate the same decoherence as a single CNOT operation. Tables 1 and 2 summarize the device parameters for our assumed eSHe quantum computer. While the numbers are still estimates, they represent a starting point from which to begin envisioning architectures for this high-mobility, low-noise QC implementation. The overall goal of this paper is to explore design trade-offs and present a computer architecture built upon these device parameters.

Noise Parameter	Value
memory decay constant	1×10^5 s
operation decay constant	5×10^3 s
transportation decay constant	2.5×10^4 s
1-qubit op error rate	1×10^{-6}
2-qubit op error rate	1×10^{-4}
measurement error rate	1×10^{-4}
reset error rate	1×10^{-6}

Table 2: Decoherence and gate noise parameters for the eSHe QC. Decoherence is modeled as an exponential decay, and gate errors are modeled as binomial probabilities.

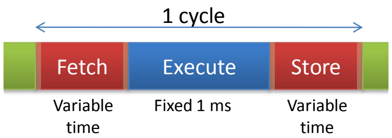


Figure 3: The execution sequence of our computer architecture consists of a sequence of variable-length cycles. Every cycle fetches, executes in a SIMD fashion, and stores a number of operand qubits. Although the SIMD operation time remains constant, the operand transportation times vary from cycle to cycle with the number of operands to transfer.

4. ARCHITECTURE AND ORGANIZATIONAL OVERVIEW OF ESHE COMPUTERS

We present a computer architecture for the eSHe QC that reflects its requirements and advantages. An eSHe QC performs 1- and 2-qubit operations in operating zones that are physically separated from memory to avoid microwave contamination. Qubit transportation is very fast relative to operation speed, which lends flexibility to microarchitecture design considerations. However, quantum algorithms are currently designed with the assumed availability of infinite parallelism, and this may still challenge the transportation network between memory and opzones if hundreds or thousands of operand qubits must be simultaneously transported for execution every cycle.

4.1 Instructions & operations

Because compiled quantum programs contain abundant data parallelism and because eSHe opzones are inherently SIMD in nature, architecture-level quantum operations should be grouped together into instruction bundles that may be executed simultaneously (independent operations execute on separate operands). The long coherence times of the eSHe qubits permits our hardware to assume a robust memory such that the qubits are expected to remain error-free for a time frame of many sequential physical operations. This allows our architecture to relegate all error correction concerns to the software level and permits a simple hardware-software interface. Section 6.4 tests the robustness of eSHe memory with a simulation-based noise model.

With this approach, software is presented a view of hardware that provides basic 1-qubit physical operations and the CNOT and CPHASE 2-qubit physical operations among arbitrary qubits. Because 2-qubit operations will dominate execution time, we will focus our attention on performing these operations efficiently. Single-qubit operations may be merged negligibly into the 2-qubit operation time with minimal overhead and, thus, we do not discuss them separately in our analysis.

The transportation of hundreds or even thousands of operands between memory and opzones is challenging and requires its own schedule in order to reduce transportation time and to avoid potential deadlock scenarios in the CCD network. Therefore, for every instruction bundle, a corresponding transportation schedule is constructed to direct the CCD network.

4.2 Execution sequence

Figure 3 illustrates the execution sequence of a cycle of operation. A transportation schedule and instruction bundle are processed every cycle leading to the following execution sequence: (1) Fetch operands from memory to execution units; (2) Operate all execution units simultaneously, and (3) Store all operands to memory.

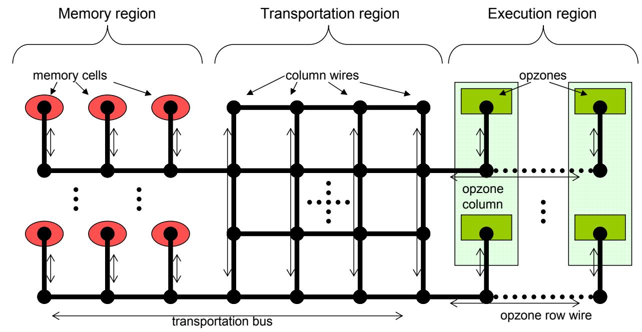


Figure 4: The Quantum Processing Unit is composed of a memory region and an execution region coupled by a transportation region. This diagram is not drawn to scale. The actual transportation region contains 9 columns, and the opzone columns are separated from each other by 100 μm. The double-headed arrows indicate CCD control. A single transportation bus signal controls all left-right movement in the memory and transportation regions. The memory and execution regions may be scaled by adding rows or columns to increase memory and execution capacity.

The operation execution time itself is constant from cycle to cycle (essentially equal to the 1 ms latency of a 2-qubit operation), but the operand transportation time may vary with the number of operands. Therefore the total execution time of the computer is a sum of the 2-qubit operation periods plus the cumulative operand transportation time. The transportation scheduler may reduce transportation costs by not storing and re-fetching operands that will be operated on during consecutive cycles.

4.3 Resulting microarchitecture

Memory, opzone, and transportation networks will be coupled together into Quantum Processing Units (QPUs) that will execute instruction bundles and their transportation schedules. For reliability and/or feasibility reasons, we anticipate that a QC might distribute its computational resources among multiple QPUs rather than a single monolithic QPU. Figure 4 shows the internal structure of one QPU. A QPU consists of a memory region where idle qubits reside, an execution region with SIMD opzone columns, and a transportation region that interconnects the memory and execution regions.

The memory region contains rows of memory cells, each separated by 10 μm distance. The memory cells are connected by individually controlled wires to east-west memory transportation rows that lead to the transportation region. The transportation region consists of a set of globally controlled east-west routes interspersed with individually controlled north-south columns. The execution region contains opzone columns with individually controlled opzone transportation rows to feed operand qubits from the transportation region. The transportation region is 100 μm wide and acts as a buffer space between the memory and execution regions, and the opzone columns are also separated from one another by 100 μm.

QPUs' execution resources (memory and opzone capacities) are likely to be varied to match application requirements. The memory capacity must accommodate the maximum number of physical qubits at any point of program execution. Likewise, the opzone capacity should match or exceed the maximum instruction bundle size in a program. The QPU's network topology design is discussed further in the following section. There, we will discuss how the QPU layout should scale with increasing execution parallelism.

5. OPERAND QUBIT TRANSPORTATION

This section presents a network construction and transportation scheduling algorithm for the Quantum Processing Unit defined in Section 4. We will show that our transportation methodology allows parallel execution capabilities (i.e., the number of opzones) to scale with only a linear increase in operand transportation time and QPU size to accommodate the higher operand traffic. Our network topology design is guided by a desire to minimize control complexity for the CCD transportation network so as to reduce the number of control signals and pins leading to the eSHe system.

5.1 QPU network topology

As described in the previous section, eSHe opzones should be distant from memory to avoid inadvertent operations on memory qubits. This constraint rules out topologies that embed opzones directly inside memory regions. The organization of the QPU network (depicted in Figure 4) has three main regions: memory, transportation, and execution. We adopt a simple two-dimensional mesh network for our QPU. The qubits are "stored" on vertices in the mesh and may travel along available links between vertices[2].

For simplicity, we have all qubits in the QPU start and end every cycle inside their memory cells. The memory region consists of rows of memory cells interlaced with memory transportation rows; each memory cell has a single, individually controlled link south to its memory transportation row. The memory transportation rows flow east/west into the transportation region, and together, the memory and transportation regions' row wires form a transportation bus that moves east or west simultaneously as described in Section 3. The purpose of the transportation region is to carry operand qubits from their memory transportation rows to their designated opzone transportation rows in the execution region. The column wires in the transportation region are individually controlled, so that qubits in each column may shift north and south independently of each other. The basic transportation operations are defined in the following subsection. Each opzone is linked to its transportation row in the same manner as the memory cells. The opzone columns and opzone transportation rows compose the execution region.

The number of rows in the QPU is exactly double the opzone column capacity so as to accommodate the opzones and their transportation rows. The memory capacity is then scaled by varying the number of columns in the memory region. The QPU's operational capacity is the product of the opzone column capacity times the number of opzone columns. Varying these two parameters for a given operational capacity adjusts the shape of the network: increasing the number of opzone columns reduces the opzone column capacity and, correspondingly, the number of rows in the network, making it both shorter and wider. We examine the impact of network shape on performance later in this section.

The goal for our transportation design methodology is to efficiently transport a large number of operand qubits from arbitrary memory cells to arbitrary opzones while minimizing CCD control complexity. Random assignment of qubits to memory cells and opzones enables simple compilation strategies, and so long as the transportation is fast, spatial locality in the QPU is unimportant. Although it is theoretically possible to individually control every single hop in the QPU network, it is desirable to reduce control complexity and minimize the number of CCD control signals. Our topology accomplishes this goal by organizing the transportation routes into wires and buses that each require only a single CCD control signal independent of the number of hops.

[2]While we refer to qubits travelling on wires and links for simplicity, recall that the qubits are actually electrons hovering *over* a mesh network of control wires.

5.2 Transportation instructions

Having defined the QPU network topology, it remains to be seen how to efficiently transport a large number of qubits. Because movement along these eSHe wires affects all qubits on that wire, each basic 1-hop movement along a wire can be viewed as a SIMD transport operation (TransportOp). Multiple TransportOps may be performed simultaneously so long as their wires do not intersect (because qubits on intersecting wires may not travel along both wires simultaneously). Here, we present the construction methodology of operand transportation schedules that are composed of bundles of parallel TransportOps. We will focus solely on the schedule that transports operands from memory to opzones; the reverse trip may be accomplished by performing this schedule in reverse. The TransportOps available to our network topology are as follows:

East: Activates the transportation bus: all rows in the memory and transportation regions shift their qubits one step to the right.

North(col): All qubits on the specified column in the transportation region shift one step upwards.

South(col): All qubits on the specified column in the transportation region shift one step downwards.

Unload(memoryCell): The qubit in the specified memory cell is shifted south into the transportation row.

Load(opZoneRow): A qubit on the easternmost transportation column is shifted right into the opzone transportation row, and all qubits in that opzone row wire are shifted right as well.

Load(opZone): A qubit is shifted north from opzone row into an opzone.

The East operation cannot be paired with any other TransportOp (except the Load(opZone)) as the transportation row bus intersects with every other wire. All the other TransportOps may be bundled together so that loading and unloading qubits can be parallelized with independent north and south movements along the transportation columns.

The selection of TransportOps is limited in order to reduce the transportation control complexity. With this selection, there is an individual link control for every memory cell and every opzone row and cell. There is a single control signal per transportation column, but the number of columns is always fixed to 9 columns (accommodating the separation distance between memory and opzones), and there is only a single remaining transportation signal that controls lateral movement for the transportation bus.

5.3 Constructing a transportation schedule

An instruction bundle specifies qubits that are operands and require transportation to their designated opzones. Given this set of operands, the transportation scheduler then organizes these operand qubits into a set of *packs*. A pack is a set of qubits that travel eastwards together out of the memory region and are always situated in the same column of the transportation region. Once a pack has reached the rightmost transportation column, it is simple to shift the pack North and South along that column wire so that its operands may be loaded into the execution region. Thus, a transportation schedule can easily be built or proven infeasible following pack assignment of the operands. The scheduling challenge is transformed into finding a successful allocation of packs that will result in a short transportation schedule. A pack assignment may result in deadlock if there is not enough slack in a pack column to permit

loading qubits into their target opzone rows. (recall that an eSHe wire may not shift a qubit beyond its endpoints)

We use a greedy search algorithm to organize the qubits into packs, and we present results for this in the next subsection. The schedule length is easily computed for each pack from its qubit elements. For each unassigned operand qubit, the algorithm finds the pack for which adding this qubit increases transportation time the least. If the desired pack already contains a qubit in that row, the algorithm finds the best alternate and conflict-free packs for these two conflicting qubits and assigns the two qubits to the packs with the lowest resulting combined transportation times. A qubit's memory cell location affects its pack schedule length, because the pack may have to delay its first Unload operation relative to other packs to avoid interfering with the composition of preceding packs. The pack search algorithm is described in Figure 5.

```
PACK ASSIGNMENT ALGORITHM
Sort all operands by shortest hopcount to their assigned opzones
For each operand qubit q
    find the pack p with the shortest schedule length for q
    if p already contains a qubit r with the same row as q then
        find alternate available packs for qubits q and r
        place q and r into those packs that minimize combined time
    else
        insert q into p
    end if
next q
```

Figure 5: Algorithm for assigning qubits to transportation packs.

5.4 Transportation design trade-offs and results

Here, we analyze how well this transportation scheduler and network approach handles increasing load. Every TransportOp involves a one-hop movement in the network and consumes 0.1 μs. Our experiments in this section construct transportation schedules for randomly generated CNOT instruction bundles and average out results from a thousand such schedules for each data point. We vary parameters including the number of opzones in the QPU, the fraction of available opzones that are used per instruction bundle, and the shape of the network.

Figure 6a shows the result of an experiment that varies the parallel execution capability (number of opzones) of a Quantum Processing Unit from 8 to 2048 opzones. For each data point, the QPU is tasked with an instruction bundle that uses all of the opzones available. The operand transportation time grows linearly as we simultaneously increase both the number of operations and the number of opzones in the QPU. Adding an operation to an instruction bundle adds about 0.5 μs for each direction of operand transport. The memory size of the QPU remains constant at 4096 bits throughout this experiment. Figure 6b shows that the size of the Processing Unit also grows linearly in this experiment. Adding opzones to a QPU results in a linear expansion of the execution region area, and the other areas of the chip may remain unchanged.

Figure 7 shows the results of a similar experiment except we keep the QPU configuration constant with 128 opzones and only vary the number of opzones in use. We again observe that the transportation time scales linearly with the number of operations. We may conclude from these experiments that for any cycle of execution, the cycle's operand transportation time is linearly proportional to the number of operations in that cycle's instruction bundle. The size of the QPU and the transportation time both scale linearly with parallel execution requirements, so neither is an immediate limitation to building larger QPUs for larger applications. Because transportation time is so much faster than operation time, we may

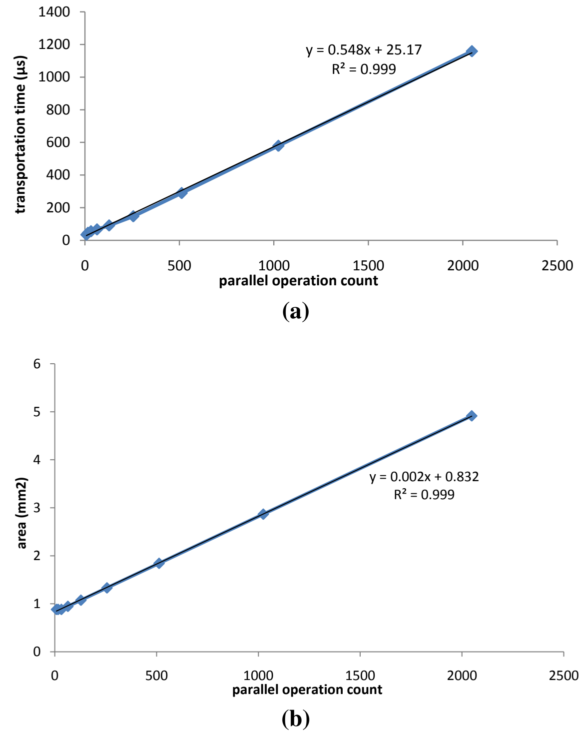


Figure 6: (a) Increasing the execution parallelism (size of instruction bundle and number of opzones in the QPU) results in linear increases in the operand transportation time. (b) The area of the QPU network increases linearly with parallelism.

transport 3554 operand qubits from memory to opzones in the time it takes to perform a single physical CNOT operation (1 ms).

The shape of the QPU network plays an important role in the operand transport time. By reducing the opzone column capacity, we may proportionally reduce the number of rows in the network making the network shorter and wider. In the previous experiments, we matched the number of opzones in the QPU with the optimal QPU shape. We now illustrate the importance of the network shape on transportation time for a specific QPU example consisting of 512 memory cells and 128 opzones under full utilization in Figure 8a. We constrained the row count to powers of 2 in order to limit the size of the search space. Figure 8b shows that as we increase the instruction bundle size beyond 128 operations, the optimal number of rows in the network remains constant at 16 rows or 8 opzones per column.

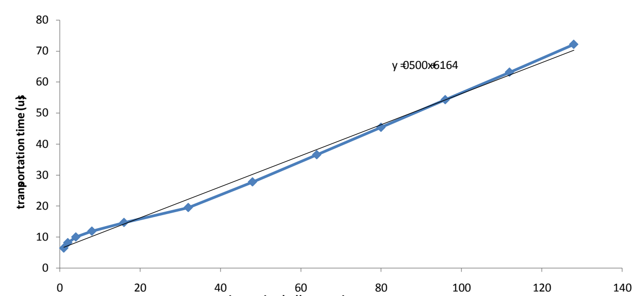
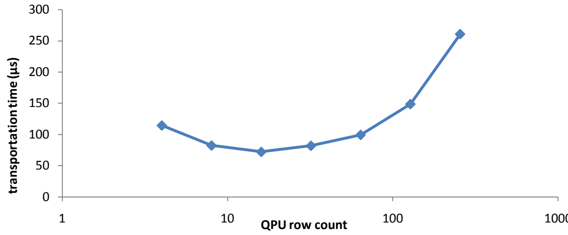
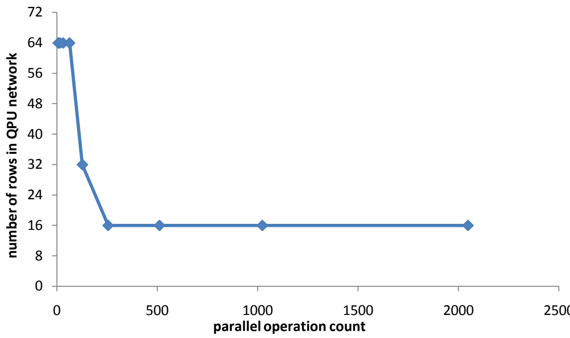


Figure 7: Varying the operation count in the instruction bundles for a fixed QPU hardware size results in transportation time linearly proportional to the number of operations.



(a)



(b)

Figure 8: (a) For any given opzone capacity, the shape of the QPU network impacts the transportation time. Increasing the opzone column capacity increases the row count of the QPU. This example with 128 opzones has an optimal network shape with 16 rows. (b) We vary the number of rows in the QPU network to vary its shape and minimize the transportation time for various levels of parallelism. The optimum row count levels off to a constant size of 16 rows as the parallelism increases past 128 total opzones.

Our transportation strategy encourages growing the number of columns because lateral movement is performed in parallel and shared by all operands. A relatively small number of rows is desired, because vertical movement is harder to parallelize. A portion of a pack's North/South movement is dependent on first offloading some members of the pack into the execution region. These vertical TransportOps cannot be executed beforehand in the pack's transportation schedule, so the easternmost pack in the transportation region is likely to require more North, South, and Loads than other packs making those TransportOps harder to parallelize. By focusing on the sharing and parallelization of the eSHe wires, our transportation strategy is able to sustain linear time and area scaling with respect to load.

6. ERROR CORRECTION AND NOISE MODEL RESULTS

A quantum computer devotes most of its computational resources (physical qubits and operations) to performing quantum error correction and protecting its encoded logical qubits from noise and decoherence. This section explores the memory longevity of eSHe qubits and evaluates the noise tolerance in the context of error recovery processes. We apply a simulation-based noise model to justify our architectural assumption of a robust memory.

6.1 Overview of the error recovery process

Logical program qubits are encoded into data code blocks so that they tolerate errors introduced by noisy gates and decoherence. Table 2 lists the best estimates for the eSHe noise parameters from Lyon [15]. Decoherence errors are modeled as an exponential decay function with the decay constant varying with the qubit's activity: idling in memory, undergoing an operation, or moving. The basic physical gate operations may also introduce errors due to precision issues. 2-qubit operations are likely to dominate as a source of errors with an estimated error rate of 1 in 10,000.

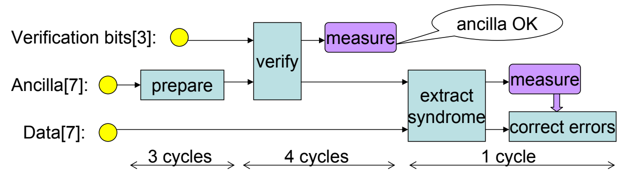


Figure 9: The simplest error recovery process: prepare the ancilla block to a specific encoded state; verify that the state is free of X errors; and extract a syndrome by interacting the ancilla with the data block. The ancilla is then measured and decoded via classical processing to determine a syndrome and the proper corrective procedure for the data block. The times listed are in terms of 2-qubit operation cycles, which are roughly 1 ms each for the eSHe QC.

The *error recovery process* identifies and corrects errors that accumulate in an encoded data block [31]. It performs this function with the aid of helper ancilla qubits that are also encoded into a code block. The ancilla block is prepared to a specific state and verified to make sure that the preparation was successful and free of errors that may propagate into the data block. A verified ancilla is interacted with the data block and then measured. This measured result is a classical bit string and is classically decoded to derive a syndrome indicating which, if any, bits of the data block contain an error. The error recovery process uses this syndrome to determine the proper corrective procedure, which is a simple 1-bit operation to the affected bits. Qubit errors may be quantized into two types: bit-flips (X errors) and phase-flips (Z errors) [16]; a qubit with both errors has a Y error. It is necessary to correct for both types of errors, so error recovery is performed twice consecutively: typically to identify and correct X and Z errors. Figure 9 illustrates the recovery procedure for the [[7,1,3]] QECC.

6.2 A simulation-based combinatorial noise model

Previous noise models used Monte Carlo simulations that statistically explore many possibilities by randomly assigning errors to qubits with a random number generator [1, 33]. The simulator measures a success if the encoded data blocks sustain fewer errors than the QECC's correction capability. This procedure is typically repeated millions of times to measure the success rate of the error recovery process. The obvious disadvantage to this approach is the time required to obtain enough samples. For example, Steane's Monte Carlo simulator [33] took days to compute data points where the recovery failure rate was only 10^{-4}. The number of samples required increases as noise rates decrease or as the error recovery process improves.

In that same paper Steane developed an alternative approach to estimate the error recovery failure rate [33]. He applied combinatorial analysis to derive formulas estimating the crash probabilities of QECCs. Our approach in this paper to modeling errors is similar but with simulation-based combinatorial analysis. The advantage of a simulation-based approach is speed and the ease of applying the noise model to different scenarios.

For each qubit, our noise model tracks the probability that it has an X, Y, or Z error. Operations and the progression of time (whether the qubit is idling, moving, or operating) increase these error probabilities. Operations also propagate errors as described in [33]. Measurements associated with verification and recovery

# syndromes	1	1	3	3
anc verif	X	X+Z	X	X+Z
crash rate	2.33×10^{-3}	1.47×10^{-3}	1.55×10^{-5}	1.54×10^{-5}
op count	28	40	84	120

Table 3: Different recovery methods yield varying crash rates and with different overheads. Crash rates (smaller is better) are shown for different recovery methods varying the number of syndromes extracted and the verification applied to the ancilla. Operational overheads for these different methods are quantified in terms of the number of 2-qubit operations involved.

processes reduce these error probabilities. We have validated our approach against a Monte Carlo simulator and have found good agreement.

6.3 Error recovery analysis

Repeating measurements is one of Preskill's laws of fault-tolerant computation [22]. This is especially important when applied to syndrome measurements in the syndrome extraction process. Determining the correct syndrome is essential for a successful error recovery; corrective measures based on an erroneous syndrome will introduce errors into the data block. One approach to improving the accuracy of the syndrome is to perform multiple syndrome extractions. By extracting three syndromes and requiring at least two of them to be consistent, the error recovery process has a higher likelihood of implementing the proper corrective procedure.

A wrong syndrome may be measured as a result of an error in one of three subprocesses: (1) the ancilla preparation procedure; (2) the ancilla verification procedure; and (3) the ancilla-data interaction (syndrome extraction). The syndrome extraction process is the least likely source of error because it involves only a single CPHASE or CNOT gate between each pair of ancilla and data qubits. The ancilla preparation procedure is relatively prone to errors because it involves 2-qubit operations among the constituent block qubits, leading to the possibility of a single gate error propagating to multiple bits. The ancilla verification procedure is meant to identify and reject ancilla blocks that contain errors. However, the verification typically only targets X errors in the ancilla, because these would be propagated into the data block [33]. This X verification is a source of possible Z errors, and these Z errors (whether from the preparation or verification steps) are the primary source of erroneous syndromes. We analyze the effectiveness of Z verification as a supplement to X verification.

Table 3 demonstrates the importance of repeated syndrome measurement for the [[7,1,3]] QECC and our assumed noise parameters. We measured the effectiveness of these error recovery approaches when applied following a logical CNOT operation between two initially error-free logical qubits. When only a single syndrome is extracted with X verification, the crash rate is 2.33×10^{-3}, which is worse than our assumed gate failure rate, indicating that error correction may not even be worthwhile with this approach. By utilizing three syndromes to guide error recovery, the crash rate falls substantially to 1.55×10^{-5}. If one of the three syndromes is wrong, recovery will still succeed if the other two syndromes do not share the same syndrome error. Although performing Z verification in addition to X verification yields a respectable improvement to the crash rate for the single-syndrome recovery process, error recovery is still not beneficial with only one syndrome extracted.

6.4 Evaluating memory robustness

A stable memory is one of our architectural assumptions based on the longevity of the eSHe qubits. In typical QC implementations, qubits accumulate enough memory noise during a single recovery process to warrant correcting every data block in the computer every recovery cycle [33]. The advantage of a stable memory is the option of not performing error recovery on idle data blocks sitting in memory; this saves precious computational resources and avoids wasting error recoveries on idle data blocks. We evaluate the robustness of the idle eSHe qubits with our noise model by introducing variable idle time into the error recovery process.

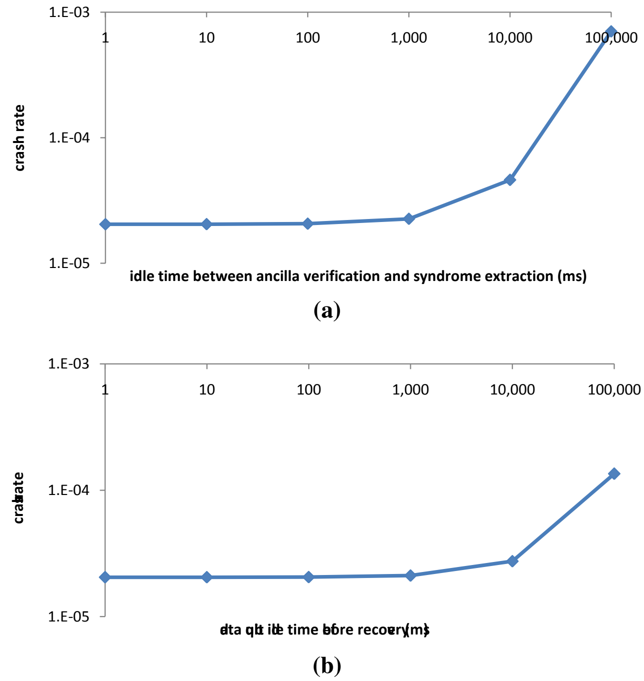


Figure 10: (a) Verified ancilla qubits may idle in memory for up to 100 ms (roughly 100 2-qubit physical operation cycles) without significantly impacting the crash rate. (b) Likewise, data qubits may idle in memory for 100-1,000 ms between successive error recoveries.

Because there is a small chance that an ancilla will fail the verification procedure (a 0.137% chance according to our model), the QECC manager is likely to produce extra ancilla blocks so that no recovery process will be starved for verified ancilla blocks. This pooling of ancilla blocks is called the *ancilla factory* approach to error correction [28]. Since qubit states decohere over time, it is important to prevent ancilla blocks in this pool from going stale. The experiment in Figure 10a introduces idle time between ancilla verification and consumption in the 3-syndrome extraction process. It shows that ancilla may idle in memory for approximately 100 ms (about 100 CNOT cycles) without significantly impacting the recovery success rate. Ancilla blocks that age for longer than 100 ms should be reverified. Figure 10b shows a similar experiment evaluating the robustness of data blocks idling in memory. Likewise, it shows that data blocks may idle in memory for 100-1,000 ms before they should undergo error recovery. These results justify our architectural assumption of a relatively stable memory that permits inactive qubits to remain idle in memory instead of undergoing constant error correction.

7. COMPILATION STRATEGIES

In this section we present compilation strategies that reduce the execution time and size requirements for an eSHe QC based on the QPU architecture that we have described. We take advantage of the QPU's support of arbitrary operations between arbitrary qubits to

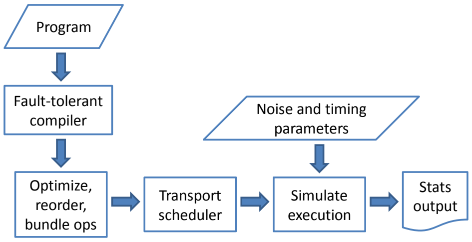


Figure 11: A flow chart of our compilation and simulation toolchain. The logical program is compiled using a fault-tolerant library incorporating a QECC and expands the program to define all the physical qubits and operations. An optimization and bundle allocation stage follows that bundles physical ops into a sequence of parallel operation bundles. The transportation scheduler builds a sequence of transport ops for every bundle. The simulator processes the compiled program and computes noise and timing results.

enable global optimizations across logical operations. We present a staggered scheduling approach that minimizes the number of cycles for any given quantum program. We also describe how a quantum program may be optimized to reduce the number of opzones required without compromising performance.

7.1 Staggering logical operations to minimize program length

Section 2 described how a quantum application is first written in terms of arbitrary quantum operations, then decomposed into fault-tolerant logical operations, and then finally translated into physical operations understood by the hardware. The goal of a compiler is to perform this software translation into hardware instructions in a manner that minimizes the execution runtime and the requisite hardware costs. Figure 11 illustrates our compilation and simulation toolchain.

Quantum programs contain data dependencies between operations just like classical programs. Because quantum operations alter their operands, operands are both sources and destinations when evaluating data dependencies. If we assume that our hardware can provide sufficient operating zones, then the program length is primarily dependent on the data dependencies forming the longest chain of operations in the program.

Quantum programs are written in terms of logical operations, and each logical operation must be expanded by the compiler to implement the appropriate encoded operation and error recovery for the selected QECC. Compilers can take advantage of libraries of optimized logical operation implementations for this purpose. We follow the fault-tolerant logical operation construction methodology defined by [22] and [32]. In this methodology, most of the physical operations are involved in the task of ancilla preparation. Ancillae are used to extract syndromes in the error recovery process as discussed in Section 6.1. A logical operation is implemented by first performing the desired encoded operation and then performing two error recoveries to cover bit- and phase-flip errors in the operands. Each recovery process uses three ancilla blocks for redundant syndrome extraction. Besides their use in syndrome extraction, ancilla qubits are also used in state preparation as part of the logical Toffoli gate.

This work focuses on two logical operations: the logical CNOT and the logical Toffoli gates. Ancilla preparation procedures dominate all logical operations. A logical CNOT operation consumes 18 cycles of execution time, and only 7 of those 18 cycles involve physical operations on the logical operand qubits. Likewise, only 10 out of 36 cycles of the logical Toffoli gate involve the operand qubits.

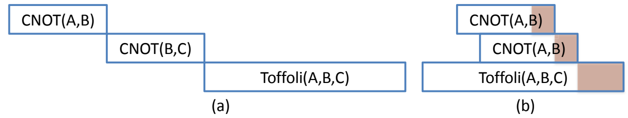


Figure 12: A sample sequence of 3 dependent logical operations.(a) A high-level compiler scheduling unstructured logical operations will not be able to overlap these operations for fear of violating data dependencies. (b) By isolating the critical physical operations (those involving the operand qubits) to the tail ends of the logical operations, logical operations may be staggered so that only their critical regions (shaded) proceed sequentially.

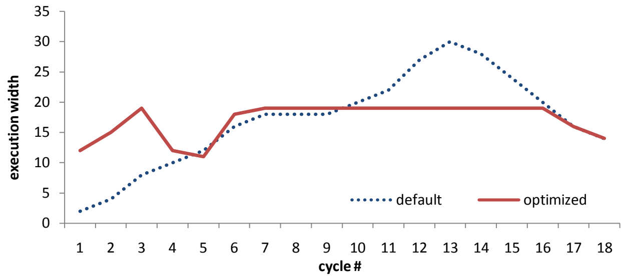
A naive approach to maintaining data dependencies may require that logical operations with data dependencies execute sequentially without overlap. Instead, we structure our logical operations in a way that enables maximal execution overlap between dependent logical operations. We sequester those few physical operations that involve the logical operand qubits to the tail end of the logical operation and designate these physical operations the *critical region* of the logical operation. These critical regions are depicted as the shaded regions in Figure 12 and contain the actual computational and recovery operations. The ancilla preparation operations precede the critical region and are independent of and may be executed concurrently with any other logical operation. A compiler may then optimally schedule a quantum program to minimize cycle time by staggering dependent logical operations so that their critical regions execute sequentially. Figure 12 illustrates the usefulness of this instruction staggering approach by completely overlapping the execution of two preceding logical CNOTs with a logical Toffoli operation.

We evaluated the effectiveness of our staggered scheduling approach versus sequential scheduling of dependent logical operations. Our sample application is a 10-bit quantum carry-lookahead adder (CLA) [7] compiled into logical CNOT and Toffoli gates using the [[7,1,3]] QECC. The CLA application is of interest because modular exponentiation, which makes repeated use of addition, is the primary runtime component of Shor's factoring algorithm. The CLA scales logarithmically with input size and is one of the most efficient addition algorithms. We find that compiling the CLA with the staggered scheduling approach yields a substantial 62% reduction in cycle time (from 342 down to 127 cycles). This speedup results from overlapping the ancilla preparation portion of each logical operation with critical computation from preceding logical operations.

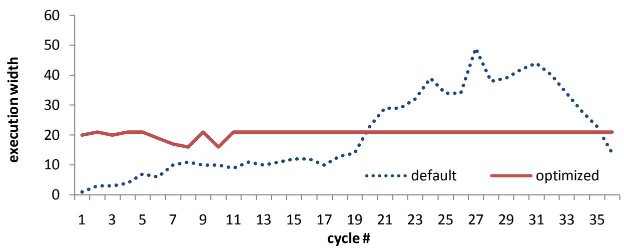
7.2 Optimizing programs to reduce execution resource requirements

Although we have shown in Section 5 that our microarchitecture can provide a high level of execution parallelism, hardware resources like opzones are still expensive. Here, we present an optimization process that reduces the peak levels of execution parallelism without increasing program length. This approach substantially reduces hardware opzone requirements without hindering performance.

We framed this optimization problem as an integer linear programming (ILP) problem and used the CPLEX software to solve it [10]. Our benchmark application is the same CLA application described in the last subsection. Our optimization process is two-fold: (1) we optimize individual logical CNOT and Toffoli operations by rescheduling their constituent physical operations; and (2) we optimize the entire program by rescheduling these blocks of optimized



(a)



(b)

Figure 13: (a) Optimizing the logical CNOT reduced the peak execution width from 30 ops to 19 ops. (b) Optimizing the logical Toffoli reduced the peak execution width from 49 ops to 21 ops.

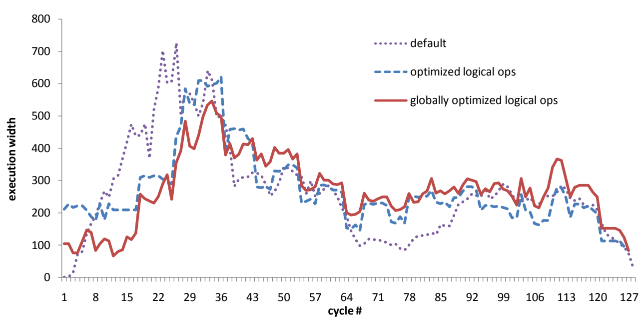


Figure 14: The 10-bit CLA program uses logical CNOT and Toffoli operations. Utilizing the optimized logical operators reduced the peak execution width from the default 726 ops to 622 ops. Globally optimizing the scheduling of these improved logical operators further reduced the peak execution width to 546 ops.

logical operations. The ILP objective in both these cases is to minimize the peak execution parallelism over all cycles. We used the LOCI (LOgical Cplex Interpreter) tool [36] to formulate the peak parallelism function: maximum instruction bundle size over a set of cycles. We defined additional ILP constraints following the example from [37] to constrain operations to a fixed window of cycles and to maintain data dependencies. Optimizing the logical operations required additional constraints to maintain the trailing critical operation regions as described in the previous subsection.

Reducing the peak execution width of a program is useful because it allows a proportional reduction in the opzone capacity and hardware size. Figures 13a and 13b show the execution width over time for the logical CNOT and Toffoli operations. With the optimized schedule, the peak execution widths of these logical operators are reduced by 37% and 57%, respectively. Figure 14 shows the execution width over time for the CLA application. Applying the optimized logical CNOT and Toffoli gates reduces the peak execution width of the CLA application by 17%. Additionally optimizing the global schedule of these logical operations within the CLA application reduces the peak execution width 25% from the default schedule.

This optimization problem further highlights the advantage in building programs out of library routines of logical operations. Individual library operations are small enough to be optimized as ILP problems scheduling their physical operations, but even a modest-sized program as our 10-bit CLA becomes infeasibly large to optimize in terms of low-level physical operations. Table 4 indicates the relative sizes of these optimization problems. The CLA program in terms of physical operations yields an ILP optimization problem file size of 126 MB. When the CLA program is expressed in terms of logical operations, the optimization problem falls down to an easier 2.75 MB, which is manageable for our computational tool set.

We have introduced compilation methods that structure logical operation blocks for simple minimization of program length as well as optimization results for reducing hardware size by reducing peak execution parallelism. These methods are applicable to any QC technology. To put transportation and operation times in context to one another for eSHe technology, we found that for the CLA application, the cumulative transportation time was roughly 37 ms compared to 127 ms of operation time.

8. RELATED WORK

Much of the previous quantum architecture work has focused on ion-trap-based QCs, and ion-trap architecture simulators have been developed for evaluating microarchitecture designs, communication schemes, and reliability [1, 17] . Balensiefer, *et al.* presented a software toolchain for the fault-tolerant compilation of quantum programs atop ion-trap hardware [1]. Their simulation framework enables the evaluation of performance and reliability, and the paper presented some initial findings on microarchitecture design and the dominant factors in reliability. Metodi *et al.* also presented a simulator framework and focused on a scalable ion-trap microarchitecture design called the Quantum Logic Array (QLA) [17]. Like our own work presented here, both of these previous works have developed software toolchains that fault-tolerantly compile quantum programs to their respective architectures and physical device models. Our work differentiates itself in several respects. Most obviously, the underlying physical implementation varies as described in Sections 2 and 3. The ion-trap QC has relatively slow mobility, whereas eSHe qubits move very quickly compared to execution time. This difference in qubit mobility has led to different architecture design decisions. For example, fine-grained tiling is an integral aspect of the ion-trap computer. The QLA's basic tiling block contains only a single logical qubit [17], and other work suggested that tiles should optimally contain only 2 ion-traps per tile [1]. By contrast, our QPUs may contain hundreds or thousands of qubits.

Communication costs are a major focus for QC implementations with less mobile qubits. Oskin *et al.* proposed the use of quantum teleportation as a long-range on-chip communication mechanism [20], and it has since become a fundamental component of many QC architecture proposals including ion-trap [17] and solid-state QCs [5]. Isailovic *et al.* detailed and evaluated an interconnection network based on quantum teleportation for ion-trap QCs [11]. They found that EPR pair distribution dominates bandwidth and network design. The availability of highly mobile qubits eliminates the need for quantum teleportation in almost all cases. Our transportation results are unique in that they focus on efficient, simultaneous transportation of a large number of operand qubits. Previous work has focused primarily on the distribution of EPR pair qubits to enable quantum teleportation.

	# qubits	# ops	# data dependencies	cycle length	LP file size
CNOT (physical)	110	307	504	18	477 KB
Toffoli (physical)	216	738	1251	36	2.37 MB
CLA (physical)	9,762	34,302	58,535	127	126 MB
CLA (logical)	36	64	126	127	2.61 MB

Table 4: The size of the optimization problem is a function primarily of the number of operations times the cycle length plus the number of data dependencies. The problem LP file is a text file defining the objective, constraints, and variable bounds and declarations for the ILP optimizer. Individual logical operations may be optimized in terms of physical operations, but the CLA program becomes too large for LOCI to process if expressed in terms of physical operations. When expressed in terms of logical operations, however, the CLA problem collapses to reasonable size for LOCI and CPLEX to handle.

Previous work has put forth the idea of dividing the QC into different regions, using a different QECC specialized for each region [4, 19, 35]. Thaker *et al.* [35] proposed implementing a memory hierarchy by providing a QC with a high-speed (albeit using a less reliable QECC) execution region fed by a cache of similarly encoded memory backed by a main memory region that uses a higher density, slower, and more reliable QECC. They found that this approach resulted in significant speedups and area reductions compared to their original QLA approach [17]. The spatial organization of logical qubits is crucial to ion-trap QCs because of their expensive qubit movement costs. However, the eSHe QC is relatively free of these locality concerns and may adopt these specializations in software rather than hardware. The eSHe QPU hardware focuses on performing physical operations and is oblivious of higher level QECC details. The eSHe software may then experiment with various QECC schemes without hardware reorganization.

Our error model assumes that noise affects each qubit randomly and independently. Traditional QECC and FT protocols handle random errors well but may not be able to defend against correlated errors that affect multiple qubits in a code block. Decoherence-free subspaces (DFS) have been proposed as an encoding layer that passively defends against correlated errors [14]. Because our physical qubits are randomly assigned memory cell and opzone locations, our code blocks (which are software data structures) are less susceptible to correlated noise than other architectures', that are more dependent on the physical locality of the constituent qubits. Nonetheless, DFS can also be applied to our architecture as an additional software layer just like QECC.

Schuchman and Vijaykumar presented a compilation technique that extracts high-level parallelism from quantum algorithms and analyzed its effectiveness in distributing coarse-grained parallel tasks to a multi-core QC [25]. They were able to obtain an approximate 2x speedup by overlapping the computation and uncomputation phases of adjacent functions in the program. Being a fairly high-level compilation technique, it may well complement our own compiler approaches as we analyze the execution of larger applications on multi-QPU eSHe QCs.

9. CONCLUSION

The eSHe QC is characterized by qubits that possess both remarkably high mobility (a range of roughly 300 m) and longevity (a memory decay constant of 100,000 s). These properties make locality unimportant and encourage the development of a new style of QC architecture. Previously proposed ion-trap and Kane QCs organize physical qubits into localized logical qubit blocks and restrict operations to local qubits in each block. These architectures rely on quantum teleportation to operate on qubits in distinct logical blocks. Our eSHe-based QPU architecture stores all the physical qubits into a single sea-of-qubits memory organization and possesses greater operational flexibility by supporting physical operations on arbitrary sets of qubits in the QPU.

We have designed a transportation infrastructure for the QPU that applies simple CCD hardware and SIMD transportation instructions to handle the qubit operand traffic necessary to feed hundreds of opzones every cycle. Our greedy transportation scheduling algorithm executes quickly and yields good communication latencies and area costs that scale linearly with load.

Memory longevity is one of the key characteristics of eSHe qubits, and it implies that idle qubits need not be error corrected every cycle. To help quantify the error recovery frequency for idle qubits, we developed a simulation-based noise model and evaluated the effectiveness of error correction protocols for qubits idling in memory. We found that the eSHe qubits may idle in memory for 100-1,000 ms without hampering the effectiveness of error correction. This result justifies the array memory architecture because qubits residing in memory are stable and require only infrequent error recoveries.

The flexibility of the QPU architecture to perform operations on arbitrary qubits translates into greater ease for the compiler in performing global optimizations between logical operations. Qubit placement within a QPU is unimportant as the transportation results have demonstrated good results with random placement. Interactions between distinct logical qubits need not be prefaced with teleportation operations. We presented compilation strategies taking advantage of these freedoms to schedule and optimize applications to minimize execution time and reduce hardware resource requirements.

In summary, the unique benefits of the eSHe QC translates into greater ease and flexibility for the hardware architecture and the software compiler. We have tested the validity and feasibility of these core characteristics by constructing a noise model and an efficient qubit transportation infrastructure.

Acknowledgements

This work was supported in part by the NSF under grant CCF-0323472, and by the ARO and DTO under contract W911NF-04-1-0398.

10. REFERENCES

[1] S. Balensiefer, L. Kregor-Stickles, and M. Oskin. An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 186–196, Washington, DC, USA, 2005. IEEE Computer Society.

[2] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and EPR channels. *Phys. Rev. Lett.*, 70(13):1895–1899, Mar 1993.

[3] C. H. Bennett, G. Brassard, S. Popescu, B. Schumacher, J. A. Smolin, and W. K. Wootters. Purification of noisy entanglement and faithful teleportation via noisy channels. *Physical Review Letters*, 76:722, 1996.

[4] D. Copsey, M. Oskin, F. T. Chong, I. Chuang, and K. Abdel-Ghaffar. Memory hierarchies for quantum data. *Non-Silicon Computing Workshop*, 2002.

[5] D. Copsey, M. Oskin, T. Metodiev, F. T. Chong, I. Chuang, and J. Kubiatowicz. The effect of communication costs in solid-state quantum architectures. In *Symposium on Parallel Architectures and Applications (SPAA) 2003*, pages 65–74, June 2003.

[6] A. J. Dahm, J. M. Goodkind, I. Karakurt, and S. Pilla. Using Electrons on Liquid Helium for Quantum Computing. *Journal of Low Temperature Physics*, 126(1-2):709–718, Jan. 2002.

[7] T. G. Draper, S. A. Kutin, E. M. Rains, and K. M. Svore. A logarithmic-depth quantum carry-lookahead adder. `http://arxiv.org/quant-ph/0406142`, 2004.

[8] M. I. Dykman, P. M. Platzman, and P. Seddighrad. Qubits with electrons on liquid helium. *Phys. Rev. B*, 67(15):155402, Apr 2003.

[9] S. Gulde, M. Riebe, G. P. T. Lancaster, C. Becher, J. Eschner, H. Häffner, F. Schmidt-Kaler, I. L. Chuang, and R. Blatt. Implementation of the Deutsch-Jozsa algorithm on an ion-trap quantum computer. *Nature*, 421:48–50, Jan. 2003.

[10] ILOG. Cplex 9.1.

[11] N. Isailovic, Y. Patel, M. Whitney, and J. Kubiatowicz. Interconnection networks for scalable quantum computers. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 366–377, Washington, DC, USA, 2006. IEEE Computer Society.

[12] B. E. Kane. A silicon-based nuclear spin quantum computer. *Nature*, 393(6681):133–137, May 1998.

[13] D. Kielpinski, C. Monroe, and D. J. Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, 417:709–711, June 2002.

[14] D. A. Lidar, I. L. Chuang, and K. B. Whaley. Decoherence-free subspaces for quantum computation. *Phys. Rev. Lett.*, 81(12):2594–2597, Sep 1998.

[15] S. A. Lyon. Spin-based quantum computing using electrons on liquid helium. *Phys. Rev. A*, 74:052338, 2006.

[16] T. S. Metodi and F. T. Chong. *Quantum Computing for Computer Architects*. Morgan & Claypool, 2006.

[17] T. S. Metodi, D. D. Thaker, A. W. Cross, F. T. Chong, and I. L. Chuang. A quantum logic array microarchitecture: Scalable quantum data movement and computation. In *International Symposium on Microarchitecture (MICRO-38)*, Barcelona, Spain, Nov. 2005.

[18] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, New York, NY, USA, 2000.

[19] M. Oskin, F. T. Chong, and I. L. Chuang. A practical architecture for reliable quantum computers. *Computer*, 35(1):79–87, 2002.

[20] M. Oskin, F. T. Chong, I. L. Chuang, and J. Kubiatowicz. Building quantum wires: the long and the short of it. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 374–387, New York, NY, USA, 2003. ACM Press.

[21] P. M. Platzman and M. I. Dykman. Quantum computing with electrons floating on liquid helium. *Science*, 284:1967–1969, 1999.

[22] J. Preskill. Reliable quantum computers. *Proc. Roy. Soc. Lond.*, A454:385–410, 1998.

[23] G. Sabouret. *Towards Spin-based Quantum Computing on Liquid Helium*. PhD thesis, Princeton University, Princeton, NJ, Jan. 2007.

[24] G. Sabouret and S. A. Lyon. Measurement of the charge transfer efficiency of electrons clocked on superfluid helium. *Appl. Phys. Lett.*, 88:254105, 2006.

[25] E. Schuchman and T. N. Vijaykumar. A program transformation and architecture support for quantum uncomputation. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 252–263, New York, NY, USA, 2006. ACM Press.

[26] P. W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26:1484, 1997.

[27] A. Steane. The ion trap quantum information processor. *Applied Physics B: Lasers and Optics*, 64(6):623–643, June 1997.

[28] A. Steane. Space, time, parallelism and noise requirements for reliable quantum computing. *Fortsch. Phys.*, 46:443–458, 1998.

[29] A. Steane, C. F. Roos, D. Stevens, A. Mundt, D. Leibfried, F. Schmidt-Kaler, and R. Blatt. Speed of ion-trap quantum-information processors. *Phys. Rev. A*, 62(4):042305, Sep 2000.

[30] A. M. Steane. Error correcting codes in quantum theory. *Phys. Rev. Lett.*, 77(5):793–797, Jul 1996.

[31] A. M. Steane. Active stabilisation, quantum computation and quantum state synthesis. *Phys. Rev. Lett.*, 78:2252–2255, 1997.

[32] A. M. Steane. Efficient fault-tolerant quantum computing. *quant-ph/9809054*, 1998.

[33] A. M. Steane. Overhead and noise threshold of fault-tolerant quantum error correction. *Phys. Rev. A 68, 042322*, 2002.

[34] A. M. Steane. How to build a 300 bit, 1 Gop quantum computer. *ArXiv Quantum Physics e-prints*, Dec. 2004.

[35] D. D. Thaker, T. S. Metodi, A. W. Cross, I. L. Chuang, and F. T. Chong. Quantum memory hierarchies: Efficient designs to match available parallelism in quantum computing. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 378–390, Washington, DC, USA, 2006. IEEE Computer Society.

[36] S.-A.-A. Touati and L. Benmouffok. Logical linear programming tool for optimizing compilation. `http://www.prism.uvsq.fr/~touati/sw/loci/`, 2005.

[37] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2000. ACM Press.